Blockchain Lab
Experiment - 1

Sanket More
D20A 29

Aim: To study cryptographic hash functions in blockchain technology and understand the working and importance of Merkle Trees.

Theory:-

1. Cryptographic Hash Functions in Blockchain

A cryptographic hash function is a mathematical algorithm that converts input data of any size into a fixed-length output known as a *hash value* or *digest*. In blockchain systems, hash functions play a critical role in maintaining data integrity, security, immutability, and trust within decentralized networks.

Widely used hash functions in blockchain include:

- **SHA-256** (Bitcoin)
- **Keccak-256 / SHA-3** (Ethereum)
- **RIPEMD-160**

These algorithms ensure that blockchain data remains tamper-proof and verifiable.

2. Characteristics of Cryptographic Hash Functions

A secure cryptographic hash function must satisfy the following properties:

1. Deterministic

The same input always produces the same hash output, ensuring consistency across all blockchain nodes.

2. Fixed Output Length

Regardless of input size, the hash output has a constant length.
Example: SHA-256 always generates a 256-bit hash.

3. Pre-image Resistance

Given a hash value, it is computationally infeasible to determine the original input, protecting transaction data from reverse engineering.

4. Second Pre-image Resistance

Given an input and its hash, it is extremely difficult to find another input that produces the same hash, preventing data replacement attacks.

5. Collision Resistance

It is highly improbable to find two different inputs that generate identical hashes, ensuring uniqueness of transactions.

6. Avalanche Effect

A small change in input produces a drastically different hash output, allowing immediate detection of data tampering.

3. Role of Hash Functions in Blockchain

3.1 Transaction Hashing

Each transaction is hashed to generate a unique Transaction ID (TXID). Any alteration to the transaction automatically changes its hash, exposing tampering.

3.2 Block Hashing

Every block contains:

- Transaction data
- Previous block hash
- Timestamp
- Nonce

These components are combined and hashed to form the block hash. Including the previous block, hash links blocks together, creating an immutable blockchain.

4. Hash Functions in Proof of Work (PoW)

In Proof of Work systems, miners repeatedly hash block data while changing the nonce until a hash satisfying the difficulty target is found:

```
Hash(Block Data + Nonce) < Target
```

This mechanism:

- Requires significant computational effort
- Prevents spam and malicious attacks
- Secures the blockchain against manipulation

5. Hash Functions in Merkle Trees

Transactions are hashed and arranged into a Merkle Tree structure. The final hash, known as the **Merkle Root**, summarizes all transactions and is stored in the block header.

Advantages:

- Efficient transaction verification
- Reduced storage requirements
- Enables Simplified Payment Verification (SPV)

What is a Merkle Tree?

A Merkle Tree (also called a Hash Tree) is a binary tree data structure where:

- Leaf nodes store hashes of individual transactions.
- Intermediate nodes store hashes of their child nodes.
- The topmost hash is called the **Merkle Root**.

Merkle Trees enable efficient and secure verification of large datasets in distributed systems such as blockchains.

Structure of a Merkle Tree

- **Leaf Nodes:** Hashes of individual transactions
- **Intermediate Nodes:** Hashes of concatenated child hashes
- **Root Node:** Merkle Root representing all data

Merkle Tree Rules

1. Each transaction is hashed individually.
2. Hashes are combined in pairs.
3. If the number of hashes is odd, the last hash is duplicated.
4. Parent hash = Hash(left child + right child).
5. The process continues until one hash remains (Merkle Root).

Working of Merkle Tree (Step-by-Step)

1. Start with a list of transactions.
2. Generate a hash for each transaction.
3. Pair adjacent hashes and hash their concatenation.
4. Repeat this process at each level.
5. Continue until the Merkle Root is obtained.

Benefits of Merkle Trees

- **Data Integrity:** Detects tampering instantly
- **Efficient Verification:** Large datasets verified using small hash proofs
- **Reduced Storage:** Only hashes are stored
- **Scalability:** Handles millions of transactions efficiently
- **Security:** Cryptographic hashing prevents forgery

Uses of Merkle Trees

- Blockchain transaction validation
- Distributed systems
- File integrity checking
- Version control systems
- Peer-to-peer (P2P) networks

Use Cases

1. Blockchain (Bitcoin, Ethereum)

Stores transaction summaries in each block and allows light clients (SPV) to verify transactions without downloading the full blockchain.

2. Cryptocurrency Mining

Merkle Root is included in block headers; changing any transaction alters the entire block hash.

3. Distributed Databases

Facilitates efficient synchronization between servers.

4. Git Version Control

Tracks file changes and ensures repository integrity.

5. Cloud Storage Systems

Verifies file authenticity without downloading complete files.

Importance of Merkle Trees

Merkle Trees enable secure, fast, and efficient verification of massive datasets, making them a foundational component of blockchain technology and distributed computing systems.

1.Write a Python program to understand SHA and Cryptography in Blockchain, Merkle root tree hash

```
import hashlib

def sha256_hash(data):
    sha = hashlib.sha256()
    sha.update(data.encode('utf-8'))
    return sha.hexdigest()

message = input("Enter a string to hash using SHA-256: ")

hash_value = sha256_hash(message)

print("\nSHA-256 Hash:")
print(hash_value)
```

```
Enter a string to hash using SHA-256: SANKET MORE D20A

SHA-256 Hash:
57cc1621a38e52a767dab695309cee014867642a8c8b9a9402b416c5db15f2ca
```

2. Target Hash Generation with Nonce: Created a program to generate a hash code by concatenating a user input string and a nonce value to simulate the mining process.

```python
import hashlib

def sha256_hash(data):
    return hashlib.sha256(data.encode('utf-8')).hexdigest()

# User input
message = input("Enter the data to be mined: ")
difficulty = int(input("Enter difficulty level: "))

target_prefix = "0" * difficulty
nonce = 0

while True:
    combined_data = message + str(nonce)
    hash_result = sha256_hash(combined_data)

    if hash_result.startswith(target_prefix):
        print("\nData        :", message)
        print("Nonce       :", nonce)
        print("Hash Result :", hash_result)
        break

    nonce += 1
```

```
Enter the data to be mined: Sanket VESIT
Enter difficulty level: 3

Data        : Sanket VESIT
Nonce       : 1018
Hash Result : 000ca0b0019f169d94ba261d1d5cc7c1c9f81f4cb189a387fcb3910f6351563e
```

3. Proof-of-Work Puzzle Solving: Implemented a program to find the nonce that, when combined with a given input string, produces a hash starting with a specified number of leading zeros.

```python
import hashlib

def proof_of_work(data, difficulty):
    nonce = 0
    target = "0" * difficulty

    while True:
        combined = data + str(nonce)
        hash_result = hashlib.sha256(combined.encode()).hexdigest()

        if hash_result.startswith(target):
            return nonce, hash_result

        nonce += 1

# User Input
data = input("Enter data: ")
difficulty = int(input("Enter difficulty (number of leading zeros): "))

# Mining Simulation
nonce, hash_result = proof_of_work(data, difficulty)

print("\nNonce found:", nonce)
print("Valid Hash:", hash_result)
```

```
Enter data: HELLO MUMBAI
Enter difficulty (number of leading zeros): 3

Nonce found: 430
Valid Hash: 000db93cd4771868d058565895ad299c61048c881b73910b14103c521417eaaf
```

```
Enter data: HELLO MUMBAI
Enter difficulty (number of leading zeros): 4

Nonce found: 50578
Valid Hash: 0000c4d6af9b473828546457ce7867be23206ab8d10187f395c744cbdc921f2b
```

4. Merkle Tree Construction: Built a Merkle Tree from a list of transactions by recursively hashing pairs of transaction hashes, doubling up last nodes if needed, and generated the Merkle Root hash for blockchain transaction integrity.

```
import hashlib

def sha256(data):
    return hashlib.sha256(data.encode()).hexdigest()

def merkle_root(transactions):

    hashes = [sha256(tx) for tx in transactions]

    while len(hashes) > 1:

        if len(hashes) % 2 != 0:
            hashes.append(hashes[-1])

        new_level = []

        for i in range(0, len(hashes), 2):
            combined_hash = sha256(hashes[i] + hashes[i + 1])
            new_level.append(combined_hash)

        hashes = new_level

    return hashes[0]

# Car-related transactions
transactions = [
    "Tx1: Ravi buys Honda City",
    "Tx2: Sneha sells Hyundai Creta",
    "Tx3: Aman books Tata Nexon",
    "Tx4: Priya purchases Maruti Swift"
]

print("Merkle Root:", merkle_root(transactions))
```

```
Merkle Root: 7a93580f4dd767d39120c704d4681da562d676ba72161b6497bc699c6acdfa2b
```

Colab Notebook Link:
https://colab.research.google.com/drive/1VL3-dD57hEeZGUWRTfEvTd8jKzD9CGLV?usp=sharing